FixD : Fault Detection, Bug Reporting, and Recoverability for Distributed Applications

Cristian Ţăpuş, David A. Noblet

California Institute of Technology
{crt,dnoblet}@cs.caltech.edu

Abstract

Model checking, logging, debugging, and checkpointing/recovery are great tools to identify bugs in small sequential programs. The direct application of these techniques to the domain of distributed applications, however, has been less effective (mostly owing to the high degree of concurrency in this context). This paper presents the design of a hybrid tool, FixD , that attempts to address the deficiencies of these tools with respect to their application to distributed systems by using a novel composition of several of these existing techniques. The authors first identify and describe the four abstract components that comprise the FixD tool, then conclude with a proposal for how existing tools can be used to implement these components.

1. Introduction

A number of techniques and tools exist to help increase and ensure reliability of small sequential software programs. In particular, the use of model checking, logging, debugging, and dynamic update techniques have historically proved to be successful in this context. However, when used in isolation, these methods do not scale well to applications in distributed environments. This is unfortunate since the extra complexity added by the high degree of concurrency present in distributed systems makes the use of such automated tools highly desirable. One of the difficulties in extending these tools for use in a distributed environment is that the processes participating in such a system only have access to local information; when a fault is encountered in a system executing in a distributed environment it is usually necessary to have information about the global state of the system in order to successfully identify what went wrong.

We propose a new system, called *FixD* , that addresses the problems of fault-detection, bug reporting, and recoverability of distributed applications. The primary purpose of our tool is to develop a substitute for the traditional *printf logging* and debugging mechanisms used extensively during the final stages of development with a more comprehensive tool that is specifically designed for distributed applications that also helps facilitate application recovery using a dynamic software update mechanism.

The main contributions of this work are:

- the specification of a generic framework for designing and testing more reliable distributed programs;
- and the design of *FixD* , which amounts to designing the glue components required to combine the various logging, debugging, and verification tools in an efficient manner;
- a model checker, called *ModelD* , that verifies safety properties embeded in OCaml programs and enables the injection of code in running programs

The rest of the paper is organized as follows: Section 2 provides an overview of existing techniques for fault-detection, bug reporting, and recovery; Section 3 gives a high-level description of the design of the *FixD* tool, identifying the four abstract components that comprise *FixD* ; Section 4 proposes an implementation of *FixD* ; the authors conclude with some suggestions for future work and a short discussion in Section 5.

2. Overview of Existing Techniques

This section provides an overview of popular techniques for fault-detection, bug reporting, and recovery.

2.1. Model checking

Model checking is an automated verification technique whereby a user specifies a model of the implementation of a system targeted for verification along

1-4244-0910-1/07/$20.00 ©2007 IEEE

with an abstract specification of the properties to which the system is required to adhere. Then, this specification is fed into a corresponding model checking tool that exhaustively explores the state space of the model, verifying that none of the legal execution paths of the system violate any of the user-specified properties.

Although model checking can be used to exhaustively explore the entire global state space of a distributed system (for this reason model checking is actually adept at discovering scheduling bugs and corner cases), the problem with this approach is that the high degree of parallelism in most distributed systems often makes this exploration infeasible. The memory required by the model checker to explore all possible schedules for the global state transitions in such a system is simply too great when there are more than a handful of processes (it is often prohibitively expensive, memory-wise, to model a moderately complex system of more than 5-10 processes).

Furthermore there are two issues that often plague the use of model checker. First, the translation of the verified abstract model to an implementation may introduce bugs. Second, it can be difficult to keep the model of a system and its actual implementation in sync (often a bug will be discovered in one, and then the fix must be implemented in both).

A partial solution to these issues is the use of model checkers that verify actual implementations. One such example is CMC [2], the C model checker, which is able to verify safety properties of C programs. This eliminates the need to create and update an abstract model if the implementation changes.

2.2. Logging and debugging

Logging mechanisms can be used to record and reconstruct a globally consistent run of the system. The collective local logs for all the entities in the system can be combined and analyzed to provide insight on the behavior of the system. Nevertheless, this reconstruction and subsequent analysis is typically laborious and is best suited to some automated tool. Furthermore, this approach relies on the availability of all the relevant logs in order to diagnose a fault.

Of course, one might not have the option to enable logging on all the entities in the system (or retrieve the logs in the event of certain failures); thus, the usefulness of this mechanism is mitigated by the control one has over the global system with respect to obtaining these logs.

An alternative to requiring the logs of all entities in the system is to record the interaction between the local component and a remote one and treat the remote entity as a black box defined only by the interaction with the local component.

The use of debugging tools in a distributed context also share the same requirements for global information as logging does, but additionally requires the ability to replay and step through the global state transitions. Some approaches to this use a distributed playback mechanism, others employ a local playback scheme; both generally make use of logging to impose a total order on all the messages sent in the system.

2.3. Logging, Checkpointing and Playback of Distributed Applications

The observation of the deficiencies of these techniques as they apply to distributed systems is not new. In fact, recently there has been much research done with the aim to solve those problems mentioned above (and others) in order to make these techniques more applicable to use with distributed systems.

One such system, implemented as a user-level library called *liblog* [1], uses logging and replay to identify bugs in distributed applications and to present the user with a trace of the distributed execution. The tool assumes though that all processes involved in the distributed computation use the logging mechanism that they provide.

Jockey [4] is another similar tool that uses binary-rewriting to detect the use of system calls and other "non-deterministic" applications and to record their interaction with the application. It provides a deterministic playback capability that enables re-running the application in the absence of the remote entities.

The Flashback tool [5] also provides a deterministic replay mechanism, but through the use of special system calls that are integrated into the Linux kernel. In order to support this playback, it intercepts system calls and logs all interactions between the target application and its environment. To reduce the overhead involved in checkpointing the state of a process, Flashback creates lightweight "shadow" processes that utilize a copy-on-write mechanism in order to avoid needlessly making copies of the memory of the original process unless the original process mutates that data. Although the Flashback tool does currently provide some support for multi-threaded programs it does not yet support distributed applications.

The techniques presented above are highly effective at providing the user with a log and a replay capability that may lead to identifying the problem that caused the failure of a distributed application.

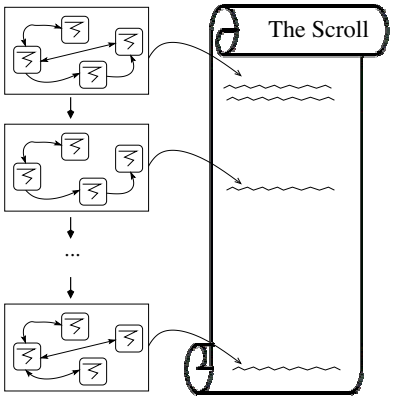

Figure 1. The Scroll. Recording the actions of the distributed components

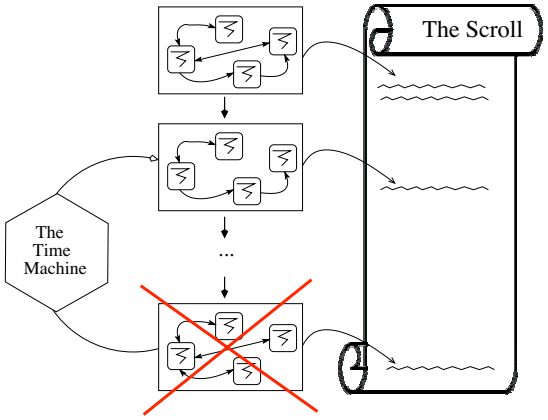

Figure 2. The Time Machine.

3. *FixD* : A Hybrid Approach

Developing, testing and debugging distributed applications can be made considerably easier if one or more of the following features are available to the programmer during application development:

- the replay of the scenario that led to an invariant violation;
- the ability to roll back the program and the related remote components to a consistent state that does not violate any invariant and investigate all execution paths that lead to an invariant violation;
- the ability to dynamically update the program when it is in a state that does not violate any invariants instead of restarting the application from the initial state; and
- the ability to use a programming model that separates the recovery code from the actual computation to make it easier to reason about the correctness of the code.

In this section we present the high-level design of our tool, *FixD* , that implements the functionality presented above. In the following text, we describe the four components that comprise the *FixD* tool and explain their interaction.

3.1. The Scroll

The first function of *FixD* is logging. Having the ability to provide the user with a clear trace of the execution of the program is useful in understanding the interaction between components of a distributed application. For this purpose we need a common *Scroll* where all or most of the components of our distributed application can record their actions and that may be used for playback or execution path investigation. It is important to notice that only nondeterministic actions (involving other components) and their outcome need to be recorded by the *Scroll* . Figure 1 tries to capture this idea by showing one application interacting with the *Scroll* at various points in its execution path.

3.2. The Time Machine

When a problem occurs and the application encounters an invariant violation we need to be able to take the entire system back to a point in time where the invariant holds, employ a mechanism to identify the conditions that led to the violation, and maybe repair the code to avoid the erroneous execution.

We call the component that provides the rollback the *Time Machine* because it provides the ability to roll back the execution of the relevant components of the distributed application to a consistent global state.

The *Time Machine* component (shown in Figure 2) provides two major functions:

- the ability to restart the execution of a distributed application from a consistent distributed state in order to investigate all possible executions and determine, through systematic analysis, execution paths that lead to invariant violations; and
- the ability to resume execution from the saved checkpoint on a different branch of execution that could bypass the error; the same applies to all related components that have rolled back.

For this to occur, we need the processes in the distributed system to interact with a checkpointing facility, in an automated and transparent fashion, such that regular checkpoints of the local state of each of

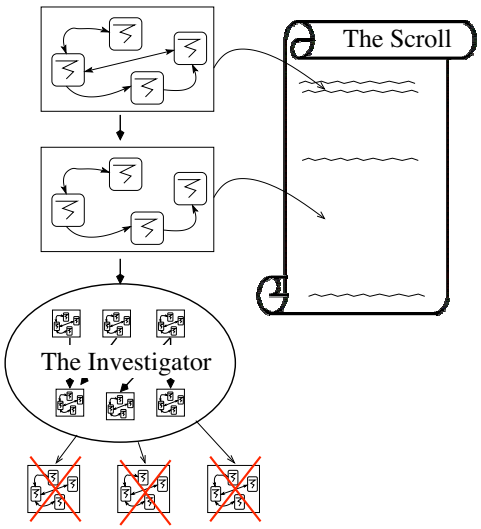

Figure 3. Exhaustively finding execution paths that lead to invariant violations.

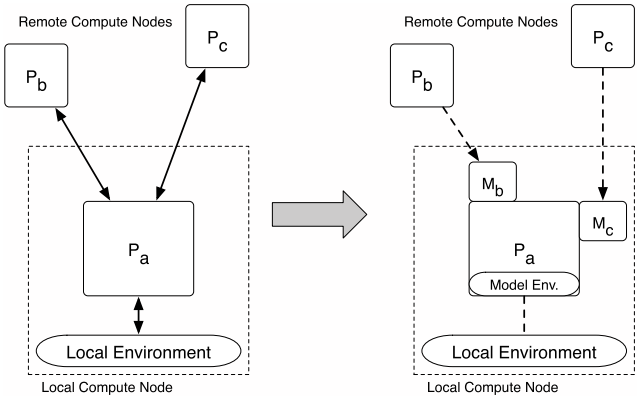

Figure 4. Response of the *FixD* mechanism during fault detection. The *Investigator* uses models for some of the external components

the processes is captured. Furthermore, communicating processes need also to agree on consistent snapshots to which to roll back their computation.

While it is not trivial to generate these globally consistent snapshots of the system based on local checkpoints (especially in an efficient manner), there do exist various techniques for doing this, one of which we describe in section 4.2.

3.3. The Investigator

If a problem is identified in an application and the state of the system is rolled back to a consistent position, one has to investigate which execution paths are the ones that led the system to an invalid state. The use of the *Scroll* can provide the user with one such execution trace (for invariant violations local to a particular process), but a more systematic approach may be needed to understand the conditions that lead to the invariant violation (i.e. there might also be a global invariant violation earlier in the system execution).

The purpose of the *Investigator* , whose interaction with other components of *FixD* is presented graphically in Figure 3, is to provide the following two functions:

- the ability to explore execution paths starting from a given state; and
- the ability to it returns a set of trails that lead to invariant violations

In the event that one process (or potentially more than one) detects a fault locally, the process that detected the fault uses the *Time Machine* component to roll back its state to a recently stored checkpoint and notifies the other processes in the system that an error has occurred. Upon receipt of this notification, each process that receives this message responds with a reply consisting of two components: a local checkpoint of the state of that process, and a model of its behavior (this model does not have to be abstract; it could simply be the implementation of the process itself); the checkpoint it provides needs to satisfy global consistency properties. The process that detected the fault initially collects these responses to piece together a consistent global checkpoint of the system that is fed to the *Investigator* component.

Figure 4 shows how, when process P_a detects a fault, the other processes send copies of their respective models to be executed locally by the *Investigator* in the P_a. It is also worth pointing out that certain parts of the environment, in this case components of the local environment itself, are not under the direct control of the *FixD* environment and must be modeled internally.

In this way, the *Investigator* component of *FixD* plays a similar role to the replay mechanism present in *liblog*. However, by being less concerned with specifically *what* went wrong and trying to reconstruct the exact conditions under which the fault occurred, this approach is better suited to determine *how* the system failed by generating a *set* of possible faulty behaviors that the system is capable of expressing.

3.4. The Healer

Once a problem has been identified in a program and problematic execution traces (i.e. those that lead to invariant violations) have been provided to the user,

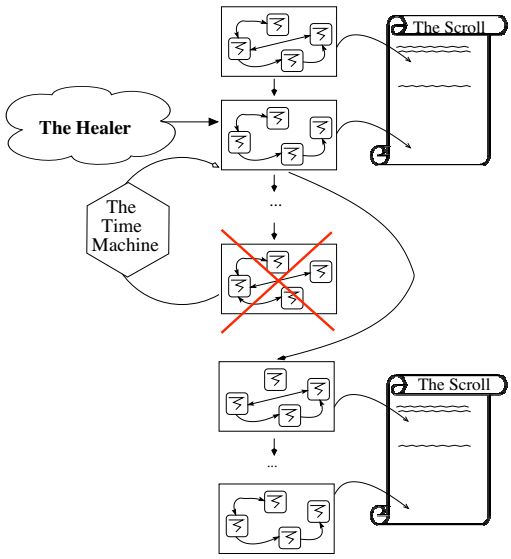

Figure 5. User intervention and dynamic updates may fix the distributed application.

the programmer can use this information to narrow down the problem in his/her code and try to provide a fix. At this point, there are several options for recovery, two of which we consider next.

One option is for the new version of the program that contains the corrected code to be restarted from the beginning. This is the simplest option and is the one that is used classically after a system failure.

Another approach, however, is for the the program to be restarted from a previously saved checkpoint where all invariants are satisfied. The advantage of this second approach is the potential to use computation that was correctly performed while executing the faulty program. This requires the ability to modify an executing process in place and provide certain guarantees that dynamically updating the process does not break type safety or invalidate any invariants. However, this might not always be possible and restarting the program from scratch could be the only option.

We call the component that provides this functionality by combining human interaction with automatic dynamic updates the *Healer* . Figure 5 shows its interaction with the rest of the components of our system.

4. Building *FixD*

One of the main contributions of this paper is identifying existing tools and suggesting methods of combining them to provide the functionality required by each of *FixD* 's components.

In this section, we describe a set of tools that we believe would be a good fit for the implementation of each of the components of *FixD* .

4.1. Implementing the *Scroll*

We identify two systems that provide the functionality required by the *Scroll* component. Their approaches are different, making each well suited for a different kind of application. The first is *liblog* [1], a user-level library that logs actions performed by distributed applications that also provides a way to replay the logs offline if necessary. The second one is the Flashback tool[5], which provides similar functionality as an extension to the Linux kernel.

The *liblog* library intercepts all *libc* calls and stores their return value in a file. Therefore, it is mainly suited for C programs. The Flashback tool, on the other hand, is more generic and it works for programs written in other programming languages, like OCaml, Java, or Fortran since it intercepts the system calls performed by the program at kernel-level.

4.2. Implementing the *Time Machine*

The *Time Machine* component can be implemented by employing a checkpointing/recovery system. In our implementation of *FixD* , we propose to use *distributed speculations* [6] as this mechanism. A speculation defines a computation that is based on an assumption whose verification may be performed in parallel with the computation. If the assumption is validated then the speculation is *committed* and the computation continues as described by the program. However, if the assumption is invalidated then the speculation is *aborted* and the process is rolled back to the state it had before entering the speculation. At this point the computation may continue on a different path of execution. The rollback is enabled by saving a lightweight checkpoint at the time the speculation is initiated.

Communication induced lightweight checkpointing using speculations. While a process executes inside a speculation it is allowed to communicate with other processes. Processes that receive speculative data are absorbed in the speculation and will have to roll back with the initiator of the speculation if the assumption the speculation is based on is invalidated.

An example of how the communication induced checkpoint mechanism used by speculations works is presented in Figure 6. Each process saves a checkpoint before receiving a new message. If process B fails as shown and it rolls back to the last saved checkpoint,

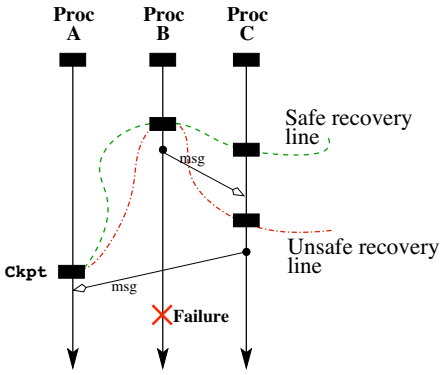

Figure 6. Safe distributed recovery lines using communication induced checkpointing.

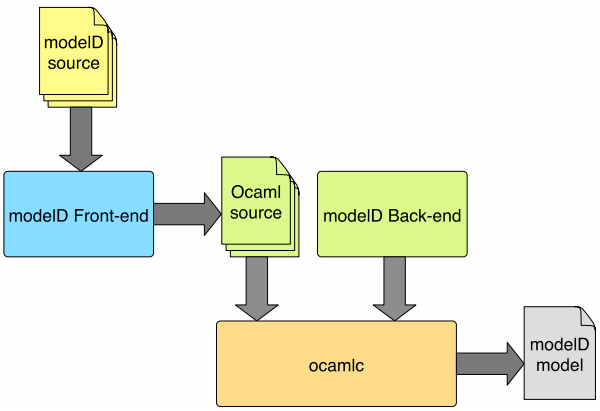

Figure 7. The components of the ModelD model checker

all other processes that communicated with it need to restore their state to form a globally consistent recovery line. The system has to prevent certain processes to roll back to checkpoints that are inconsistent with the rest of the system. By using speculations this information is automatically determined by the system and each process rolls back to the correct checkpoint, after which the model checking component takes over, as described before.

There are two main differences between speculations and traditional checkpoint and rollback mechanisms, as follows:

- Speculations use a copy-on-write mechanism to build lightweight, incremental checkpoints of processes.
- Speculations allow applications to use a different execution path upon rollback, depending on how the assumption was invalidated.

These differences have the advantage that (1) checkpoints generated using speculations introduce less overhead than certain types of traditional checkpointing, and (2) the alternate execution path used in case of a rollback may be used to repair the problem that led to the failure of the application in the first place. This makes the choice of using distributed speculations particularly attractive for the implementation of the *Time Machine* component of *FixD*

4.3. Implementing the *Investigator*

The *Investigator* functions very much like a traditional model checker. The main difference is that we want to be able to exhaustively analyze the behavior of real programs rather than that of abstract models.

Implementing a model checker for real programs would have to depend on the language that the application was implemented in and would require access to the source code in order to provide meaningful traces to the programmer. Therefore, we suggest two such systems that provide the desired functionality. One is *ModelD* , a model checker of OCAML programs that is part of the contributions of this work. The second one is CMC [2], a model checker for C programs. Both are discussed below.

ModelD: modeling OCAML programs. The implementation of the ModelD model checker consists of two primary components (see Figure 7): a back-end model checking engine used to perform model checking on real Ocaml code, and a front-end syntax extension to the Ocaml grammar (written using Camlp4) that is used to provide a convenient interface for the user to interact with the back-end engine (so that the resulting language is more like a conventional model checking language).The back-end component is responsible for performing the actual state transitions, keeping track of the visited execution paths (calculating the reachability graph), and verifying that no user-specified invariants are violated. The model checking engine is based on a guarded command model, where the behavior of the system is described by a set of guarded commands that can be chosen for execution any time.

Although our proposal of the design of *FixD* does not rely on the facilities of any particular model checker, the features of ModelD (and especially the back-end component) present a good fit for this application. The back-end component of ModelD is highly flexible and supports the ability to dynamically change the set of actions available to the model checking engine as well as the ability to customize the search order for the state graph. Though these mechanisms were

originally introduced in the ModelD engine as a way to support heuristic search, they can be applied in a less conventional context in order to allow the ModelD engine to run the actual implementation of a process involved in a distributed application.

As an example, consider the implementation of an event-based protocol where the events in the system are mapped to actions in ModelD (where the guards for the ModelD actions are predicates that indicate when an event would be triggered). Furthermore, suppose all communication is handled by other generic actions (perhaps provided via some library of such actions included as part of *FixD*). In this way, each event is a state transition within the model checker, and, given that we have the ability to control the search order of the states within the model checking engine, we can ensure that we only pursue a single execution path (the path the "conventional" implementation would take). In the event that the process detects a fault, we can simply restore one of our previous states, swap out the real communication actions, replace those with *models* of the communication actions, and add the actions that correspond to the behavior of the other processes in the system (we would get these from the other processes themselves, as described in the section above).

CMC: The C Model Checker. CMC [2] is a model checker that generates the state space of a given application by executing the C or C++ source code. During the state space exploration, CMC automatically checks for certain generic properties such as memory leaks and invalid memory accesses. Also, CMC reports any deadlock states, in which the system can make no progress. To check for specific properties, the user has to provide additional invariants (written in C). CMC evaluates these invariants in every state it generates.

Implementation Abstractions. Whenever model checking is employed, there is always a concern that there could be a mismatch between the behavior of the model being used and the actual system the model is supposed to represent. Although this is always a problem to some degree with any model checking solution, there are approaches to model checking that are designed to minimize the effect of this. In particular, we envision the use of a model checking tool that is designed to work directly on an implementation language without having to modify the implementation of the system. This way, there is less of a chance of the model differing from the actual system itself. Of course, there will always be components of the system that will be outside the control of the *FixD* environment (such as the network itself, in the case of communicating processes); in the case of such components it may be necessary to have abstract models of their behavior, but perhaps many of these could be formally verified and included as part of the *FixD* tool itself.

4.4. Implementing the *Healer*

The *Healer* is a complex component that also involves a significant amount of involvement from the programmer. Once a bug is identified in the code and the *Investigator* provides execution traces of possible invariant violations, the programmer repairs the source code of the program and, as mentioned before, either restarts the program or resumes execution from a stable checkpoint by dynamically updating the executable with the corrected code. We discuss two such tools that use the second approach of dynamically updating the application. They are aimed at two different source languages. One is the ModelD, which was also used to implement the functionality of the *Investigator* . The other one is called Ginseng[3], a dynamic software update tool for C programs.

ModelD. The back-end component of ModelD allows one to inject actions that divert the execution of a program using an updated version of the actions. This is performed in a manner similar to that described in Section 4.3. Internally, this operation simply reduces to dynamically changing the set of enabled actions. However, additional steps need to be taken in order to ensure that a state in the original implementation is equivalent to some resulting state in the updated implementation. Therefore, the programmer has to either force rollback to a point were this condition can be automatically verified or has to write the update such that state equivalence is guaranteed.

Ginseng: Dynamic Software Updating for C. Ginseng has three components: a compiler, a patch generator and a runtime system for building updateable software. The main functionality is provided by a static analysis tool that also makes it one of the most safe systems of its kind. The main properties of the system, which make it practical are: (1) it does not require extensive changes to the application; (2) it restricts the form of the dynamic updates as little as possible; and (3) it makes significant headway toward making updates easy to write and easy to establish as correct.

		preventive	diagnostic	treatment	comprehensive	opportunistic
Techniques	Model Checking (MC)	√	–	–	√	–
	Logging (L)	–	√	–	–	√
	Checkpoint & Rollback (CR)	–	–	–	–	√
	Dynamic Updates (DU)	–	–	√	–	–
	Speculations (S)	–	–	√	–	√
Tools	liblog (L & CR)	–	√	–	–	√
	CMC (MC)	–	–	–	–	√
	FixD (M & L & S & DU)	√	√	√	√	√

Figure 8. The characteristics of the techniques and tools discussed in this paper. The abbreviations of the first five mechanisms are used to define what the tools use to provide their functionality.

4.5. Future Work

In order to facilitate the development of the proposed *FixD* tool, it would be useful to leverage the availability of existing work in the field. Thus, it would be advantageous to first improve on previous projects we have developed (namely ModelD and the kernel level distributed speculations) and then integrate some of the work that others have done.

One such improvement is to develop a set of general-purpose models designed to integrate with ModelD in order to imitate the behavior of common and well-known components of the environment of a process. For example, it would be useful to have models of various components such as network communication or disk access and perhaps many of the available system calls. This way, it would not be necessary for each process to model these elements of the environment directly.

We also plan to investigate the use of speculations in conjunction with dynamic libraries to enable runtime code instrumentation for automatic error recovery and for automatic runtime code updates. Such work could allow for *FixD* to provide some mechanism for dynamic fault recovery instead of simply allowing for detection and reporting.

Moreover, there are other tools that would be appropriate components for the development of *FixD* . In particular, it would be interesting to explore the benefits of using another implementation-language model checker like CMC.

5. Discussion

In this paper we present the design of a novel tool for fault detection, bug reporting, and recoverability that is targeted at distributed applications. We discuss several other technologies that attempt to address the same general problem. Figure 8 presents an overview of the characteristics of the techniques and tools discussed in this paper, both from the point of view of the type of service they provide (preventive, diagnostic, or treatment) to find and cure bugs, and of the generality of the service (comprehensive or just opportunistic). Our *FixD* system, while still in its initial stages of development, is a promising tool that combines a wide spectrum of techniques in order to create a more complete debugging/recovery utility.

References

[1] D. Geels, G. Altekar, S. Shenker, and I. Stoica. Replay debugging for distributed applications. In *USENIX Annual Technical Conference 2006*, 2006. 2.3, 4.1

[2] M. Musuvathi, D. Park, A. Chou, D. R. Engler, and D. L. Dill. CMC: A Pragmatic Approach to Model Checking Real Code. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, Dec. 2002. 2.1, 4.3, 4.3

[3] I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol. Practical dynamic software updating for C. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, June 2006. 4.4

[4] Y. Saito. Jockey: a user-space library for record-replay debugging. In *AADEBUG'05: Proceedings of the sixth international symposium on Automated analysis-driven debugging*, pages 69–76, New York, NY, USA, 2005. ACM Press. 2.3

[5] S. Srinivasan, S. Kandula, C. Andrews, and Y. Zhou. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In *USENIX Annual Technical Conference, 2004.*, pages 29–44, 2004. 2.3, 4.1

[6] C. Ţăpuş. *Distributed Speculations: Providing Fault-tolerance and Improving Performance*. PhD thesis, California Institute of Technology, Pasadena, CA, June 2006. 4.2